Etape 11 : Tout rendre exécutable *Runnable* et *Callable*

Multithreading en JAVA

Rendre ses objets exécutables

- on a vu la création d'objets exécutables à partir de la classe Thread
  - Il s'agit d'objets conçus pour être exécutables mais peut-on donner cette capacité à d'autres objets ?
  - Réponse OUI, les interfaces sont faites pour ça :
    - *Runnable =>run*
    - *Callable => call*

Runnable

- historiquement, c'est la première (Java1.1)
- *Thread implémente Runnable => on la connait déjà un peu*
  - *méthode run à implémenter :*
    - *void run() { // ce qui doit être exécuté }*
- *MAIS*
  - *=> l'objet Runnable ne bénéficie pas du start() pour être lancé*
  - *=> a besoin d'un véhicule technique : un thread*
- pour lancer un Runnable on le placera dans un Thread et on appellera start()

Runnable

```
class MonImplDeRunnable implements Runnable {
    public void run() {
        for (int i = 0; i < 5; i++)
            System.out.println(i);
    }

    public static void main(String[] args) {
        Thread vehicule;
        MonImplDeRunnable objExec = new MonImplDeRunnable();
        vehicule = new Thread(objExec);
        vehicule.start();
    }
}
```

Tout rendre Exécutable

- L'intérêt d'une interface se situe dans le transtypage multiple :

```
class Etudiant extends Personne implements Runnable {

public void run() {
    While (true) {
                this.travailler();
            }
    }

}
```

  - Utile notamment pour les interfaces graphiques … / …

Tout rendre Runnable

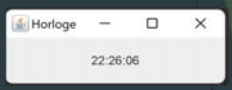
```
public class Horloge extends Label implements Runnable {

    private DateFormat timeFormat = DateFormat.getTimeInstance();

    public void run() {
        try {
            while (true) {
                this.setText(timeFormat.format(new Date()));
                Thread.sleep(1000);
            }
    } catch (Exception exception) {
        exception.printStackTrace();
        }
    }

    public Horloge() {
            this.setText(timeFormat.format(new Date()));
            this.setAlignment(Label.CENTER);

    // Création d'un thread basé sur l'horloge créée.
            (new Thread(this)).start();
    }
```



Les objets *Callable*

- Cousine de Runnable mais historiquement postérieure (Java 1.5) dans le package *java.util.concurrent*
- au lieu de *run* il faut implémenter *call*

- Alors … pourquoi l'ajouter ?
  - Donne la possibilité de
    - retourner une valeur à la fin de leur exécution
    - faire suivre une exception à l'appelant

Les objets *Callable*

- `public interface Callable<V>`
  - V sera le type de la valeur retournée à la fin de l'exécution du thread

- `V call() throws Exception`
  - à comapraer au *void run() de Runnable :*
    - *on a un type de retour et des exceptions retransmises*

Exécuter un *Callable*

- A la différence de sa cousine Runnable les callable ne peuvent pas être directement embarqués dans un thread
- Alors comment on véhicule un Callable ?
  - On les confie à ceux qui savent … exécuter : les Executor
  - Les Executor (interface) arrivent eux aussi avec l'api 1.5
- Second avantage de l'api concurrent : on prend de la hauteur en déléguant l'exécution à des spécialistes

Exécuter un callable

- Executor
  - Interface qui représente un objet qui exécute une tâche (Runnable ou Callable)
  - Selon l'implémentation les modalités d'exécution seront très différentes
  - => Découplage de tâche et de son exécution
- ```
  public interface Executor {
              void execute(Runnable var1); }
  ```
  ```

Bien choisir son mode d'exécution

- ExecutorService
  - Permet de demander l'exécution d'un Callable ou d'un Runnable
  - On ne crée pas les thread soi-même c'est l'instance d' ExecutorService qui s'en charge et lui attribue le callable ou le runnable
- Plusieurs modalités d'exécution sont offertes

Bien choisir son mode d'exécution

- On veut un thread unique pour notre tache :
  - choisir : Executors.newSingleThreadExecutor(),
- Besoin d'un pool de thread réutilisables pour un ensemble de tâches :
  - Vous aimerez notre Executors.newFixedThreadPool(int nThreads),
- On veut beaucoup de performance avec de nombreuses tâches courtes :
  - Executors.newCachedThreadPool() est fait pour vous,
- Besoin de programmer à l'avance ou répéter l'exécution de votre tâche
  - Connaissez-vous l' Executors.newScheduled….(…)

Soumettre l'exécution du Callable

- ExecutorService
  - Ensuite on utilise une méthode *submit* pour démarrer le Callable dans le service d'exécution
  - Submit n'attend pas la fin de l'exécution (on est en asynchrone) mais le callable va déposer son résultat dans un objet sépcial pour les futurs utilisateurs de ce résultat

Bien finir l'exécution

- ExecutorService
  - Une fois le service démarré il attend des demandes d'exécution (submit)
  - Il faut l'arrêter
- *shutdown(),* demande à l'Executor de s'arrêter lorsqu'il aura terminer ses tâches en cours
- *shutdownNow()* stoppe l'Executor immédiatement, quel que soit l'état d'avancement des tâches en cours

Retour pour le Future

- Les objets retournés par les Callable ont été standardisés afin de faciliter leur exploitation par thread appelant
  - Il s'agit du type Future (car il s'agit essentiellement du contenu qui SERA retourné par l'exécution)
- `public Interface Future<V>`
- Le submit demande à l'Executor de remplir le Future dès qu'il sera disponible

Ce que l'on peut savoir sur le Future

- L'interface propose plusieurs méthodes (avec des varaintes) qui renseignent sur le résultat attendu et l'état de la tache *Callable* :
- `V get()`
  - afin de récupérer le résultat : on attend si pas dispo
  - on peut spécifier un délai max d'attente
- `Boolean cancel()`
  - Annuler la tâche
- `Boolean isCanceled()`
  - Est-ce que la tâche a été annulée ?
- `Boolean isDone()`
  - Est-ce que la tâche est finie ?